# Split-Radix Fast Fourier Transform Using Streaming SIMD Extensions

## Version 2.1

## 01/99

Order Number: 243642-002

# Table of Contents

## Revision History

| Revision | Revision History | Date |
|---|---|---|
| 1.0 | Original publication of document | 3/98 |
| 2.0 | Verified simulation results | 7/98 |

## References


1 *Introduction to Digital Signal Processing*, Proakis and Manolakis, (Macmillan, 1988, ISBN 0-02-396810-9)

2 *Advanced Digital Signal Processing*, Proakis, Rader, Ling and Nikias, (Macmillan, 1992, ISBN 0-02-396841-9)

3 *The Fast Fourier Transform and Its Applications*, Brigham, E. Oren, (Prentice-Hall, Inc., 1988)

4 *The Fourier Transform and Its Applications*, Bracewell, Ron N., (McGraw-Hill Book Company, 1965)

5 *The analysis and restoration of astronomical data via the fast Fourier transform*, Brault, J. W. and White, O. R., 1971, Astronomy & Astrophysics., 13, pp. 169-189.

6 *Split-radix FFT Algorithm*, Duhamel, P. and Hollmann, H., 1984, Electr. Letters, vol. 1, pp. 14-16, January.

7 *An algorithm for the machine calculation of complex Fourier series*, Cooley, J. W. and Tukey, J. W., 1965, Mathematics of Computation, 19, 90, pp. 297-301.

8 *Digital Signal Processing*, Oppenheim, Alan & Schafer, Ronald, (1975, ISBN 0-13-214635-5)

9 *Implementation of a High-Quality Dolby* Digital Decoder Using MMX™ Technology*, Abel, J. and Julier, M., 1997, Intel Technology Journal Q3 1997, http://developer.intel.com/technology/itj/q31997/articles/art_3.htm

Additional Information:

10 *An Introduction to Fourier Theory*, by Forrest Hoffman, ("http://aurora.phys.utk.edu/~forrest/papers/fourier/index.html#introduction"):

# 1 Introduction

Streaming SIMD Extensions for the Intel® Architecture (IA) instruction set provides floating-point single-instruction, multiple-data (SIMD) instructions. These single-precision floating-point instructions provide a means to accelerate operations typical of 3D graphics, real-time physics, and spatial (3D) audio. This application note discusses the implementation of an FFT algorithm using Streaming SIMD Extensions, and presents code examples that exploit these instructions.

Linear transforms, especially Fourier and Laplace transforms, are widely used in solving problems in science and engineering. The Fourier transform is used in linear systems analysis, antenna studies, optics, random process modeling, probability theory, quantum physics, boundary-value problems (Brigham, 2-3), radar, sonar, MPEG audio compression, Dolby Digital audio compression, spectral analysis and restoration of astronomical data (Brault and White). The Fourier transform, a pervasive and versatile tool, is used in many fields of science as a mathematical or physical tool to alter a problem into one that can be more easily solved. Some scientists understand Fourier theory as a physical phenomenon, not simply as a mathematical tool. In some branches of science, the Fourier transform of one function may yield another physical function (Bracewell, 1-2) (Forrest Hoffman).

## 1.1 The Fourier Transform

The Fourier transform, in essence, decomposes or separates a waveform or function into sinusoids of different frequency which sum to the original waveform. It identifies or distinguishes the different frequency sinusoids and their respective amplitudes (Brigham, 4). The Fourier transform of $x(t)$ is defined as:

$$X(F) = \int_{-\infty}^{\infty} x(t)\, e^{-j2\pi Ft}\, dt. \qquad \text{[equation 4.1.24 in Proakis and Manolakis]}$$

The inverse Fourier transform (also known as the synthesis equation) is:

$$x(t) = \int_{-\infty}^{\infty} X(F)\, e^{j2\pi Ft}\, dF. \qquad \text{[equation 4.1.29 in Proakis and Manolakis]}$$

In digital systems, the input is usually sampled at discrete time intervals. The Fourier transform for discrete-time signals is:

$$X(\omega) = \sum_{n=-\infty}^{\infty} x(n)\, e^{-jwn} \qquad \text{[equation 4.2.30 in Proakis and Manolakis]}$$

The inverse Fourier transform (also known as the synthesis equation) for discrete time signals is:

$$x(n) = 1/2\pi \int_{2\pi} X(\omega)\, e^{j\omega n}\, d\omega.$$ [equation 4.1.35 in Proakis and Manolakis]

Most real-world signal processing applications operate on a finite data set (*i.e.* a finite number of samples or frequency bins). For finite-duration sequences, the Discrete Fourier Transform (DFT) is:

$$X(k) = \sum_{n=0}^{N-1} x(n)\, e^{-j2\pi kn/N} \qquad k = 0, 1, 2, \ldots, N-1$$ [equation 4.5.46 in Proakis and Manolakis]

where N is the number of points (*i.e.* complex values) in the array. For finite-duration sequences, the Inverse Discrete Fourier Transform (IDFT) is:

$$x(n) = 1/N \sum_{n=0}^{N-1} X(k)\, e^{j2\pi kn/N} \qquad n = 0, 1, 2, \ldots, N-1$$ [equation 4.5.47 in Proakis and Manolakis]

## 1.2 Data Types

A 32-bit floating-point representation was used in this implementation. One alternative would be to use a 16-bit integer representation, or a mixed 16/32-bit integer representation, as would be possible using MMX™ technology. For applications that have a large dynamic range, or stringent signal-to-noise requirements, this 32-bit floating-point implementation may have a compelling increase in quality. An example of this is high-quality audio.

## 1.3 Fast Fourier Transform

The Fast Fourier Transform (FFT) is a DFT algorithm developed by Tukey and Cooley in 1965 which reduces the number of computations from $O(N^2)$ to $O(N * \log_2 N)$[1]. The restriction is that N must be a power of 2. This FFT is known as radix-2. Additionally, if N is a power of 4, a radix-4 FFT can be used. There are many texts that cover the FFT algorithm, such as Proakis and Manolakis, and Oppenheim and Schafer.

While the numerical result of the FFT is identical to that of the DFT, the ordering of the output is different. This is known as bit-reversed addressing order. One can create the bit-reversed

[1] For those unfamiliar with this notation, O(X) represents "Complexity of Order X". For example, an algorithm of O(X) has a computational requirement that increases linearly with increasing X, whereas a an algorithm of $O(X^2)$ increases exponentially with increasing X. This is an approximation of the computational requirements of various algorithms, and can be used to compare the algorithms.

addressing order by representing the address as a binary number of length $\log_2(N)$ and doing a 'mirror image' on the bits. For example:

| Normal order | | Bit-reversed order |
|---|---|---|
| 000 | -> | 000 |
| 001 | -> | 100 |
| 010 | -> | 010 |
| 011 | -> | 110 |
| 100 | -> | 001 |
| 101 | -> | 101 |
| 110 | -> | 011 |
| 111 | -> | 111 |

For another example, see Figure 2 for a bit-reversed output.

# 2 Split-Radix FFT Algorithm

In an effort to create an efficient FFT, the Split-Radix FFT algorithm [Duhamel and Hollmann (1984), and covered in Proakis, Rader, Ling and Nikias] was used. It has the following characteristics:

- Uses the Split-Radix technique
- Uses single-precision (32 bit) floating point number representation
- Uses complex (real and imaginary) number representation for both input and output arrays
- Uses an in-place transform (output is placed in the input array)
- Output is in bit-reversed addressing order

The Split-Radix FFT algorithm has been optimized for the Katmai processor using SIMD floating point instructions. This algorithm was selected because it uses a lower number of operations compared to Radix-2 and Radix-4 implementations. Table 1 compares the number of complex multiplications for common FFT implementations.

**Table 1: Complex Multiplication Comparison**

| Implementation | Number of Complex Multiplications |
|---|---|
| Radix-2 | $1/2\ N \log_2(N)$ |
| Radix-4 | $5/8\ N \log_2(N)$ |
| Split-Radix | $1/3\ N \log_2(N)$ |

The data given in Table 1 does not factor out trivial multiplications (e.g. by 1.0 or -1.0), and does not cover complex additions and subtractions. However, since complex multipications are very processor-intensive (requiring four real multiplications, one real addition and one real subtraction), these values can be used as an approximation of the relative complexity of the final implementation.

## 2.1 The Function of the FFT Algorithm

The basic element of the split-radix FFT is the L-Butterfly, as illustrated in Figure 1.

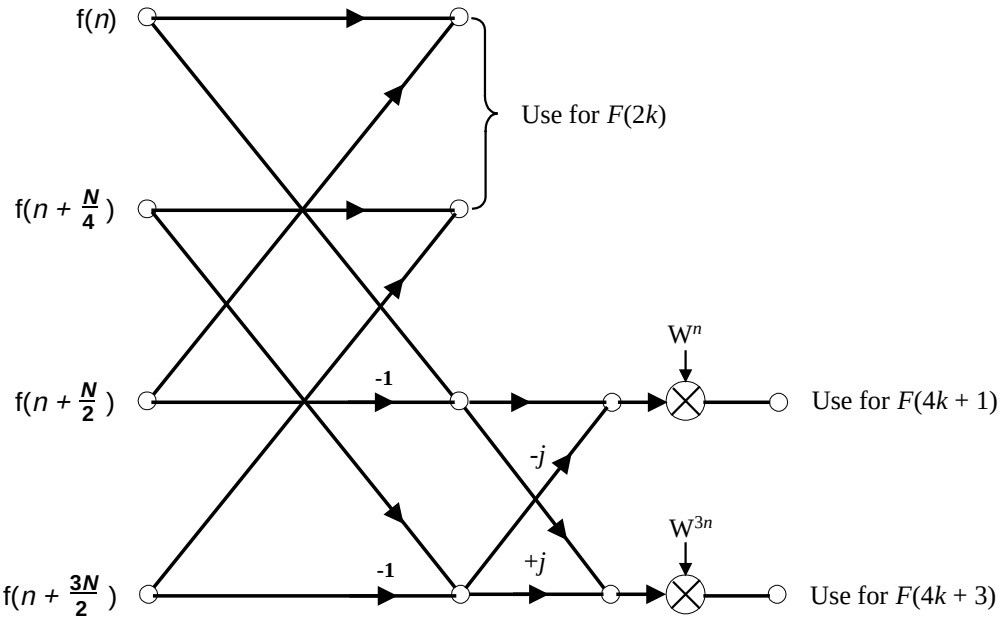


**Figure 1: FFT L-Butterfly**

Figure 2 shows the entire algorithm flow for one 16-point FFT.

Where:

$W^{kn} = e^{-j2\pi kn/N}$

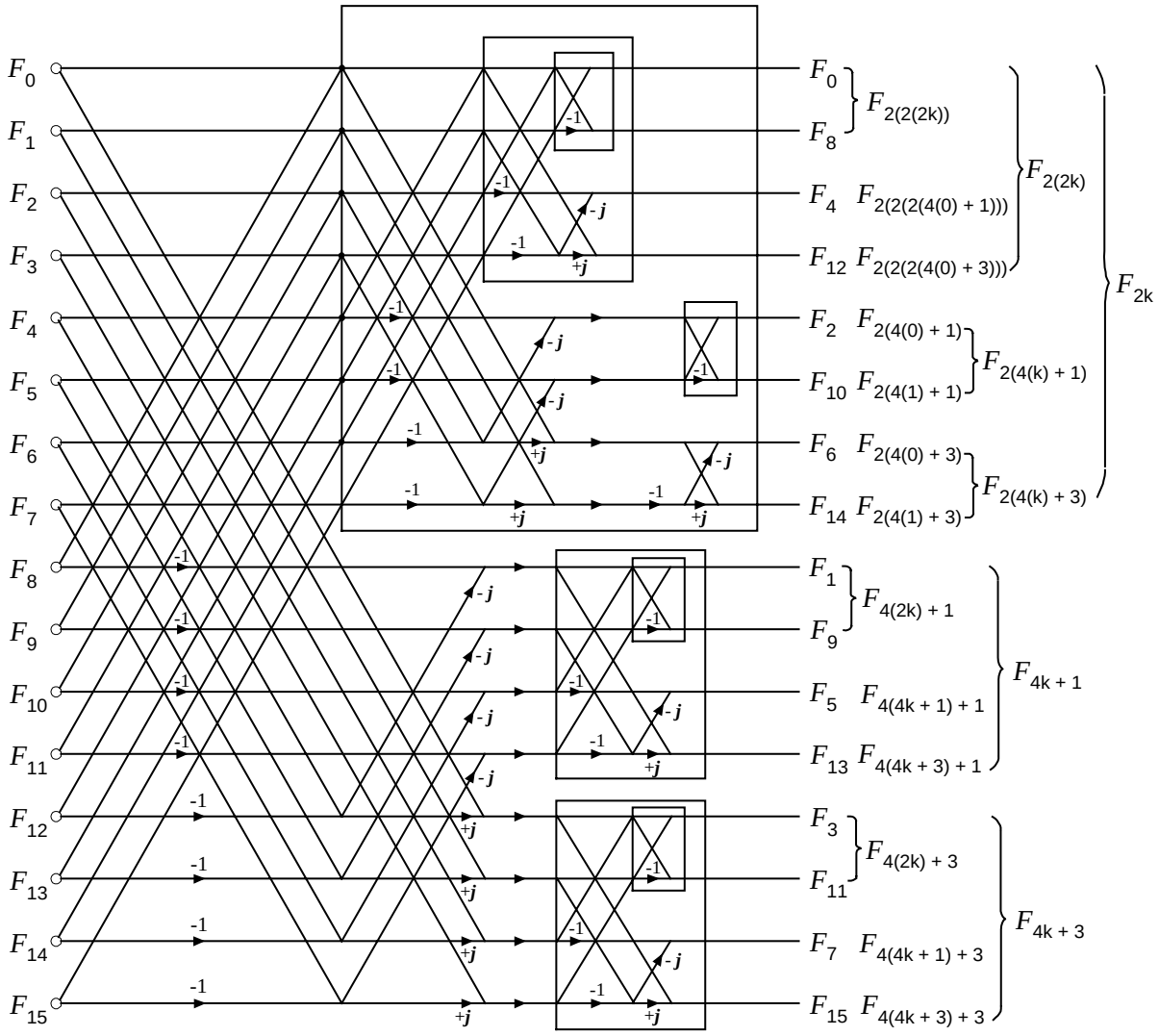


**Figure 2: One Entire 16-Point Split Radix FFT**

The algorithm has a recursive nature (see Figure 1). This means that the algorithm is first performed as an N-point L-butterfly, and then the results are fed to one N/2-point and two N/4-point stages, that themselves can be performed as an L-butterfly. This continues until all the points have been fully transformed. Note that near the end of the process, 8-point and 4-point transforms using the L-butterfly yield 2-point outputs. For 2-point transforms, simple 2-point FFT butterflies are performed.

The sequence of execution is as follows: In the first step of the execution, the first set of L-butterflies are all calculated, yielding the outputs of all the L-butterflies from the original input data. The number of L-butterflies calculated is one-fourth of the size of the data set. These outputs are logically combined into three sections, one that has one-half the data points, and two that have one-fourth the data points each. Then each of the three sets of data are transformed by independent calls that perform split-radix FFTs, each of them creating three more data sets that need to be transformed. This continues until a small enough data set is encountered. In the diagrams, these are 2-input DFTs, but for efficiency these are combined with larger transforms.

## 2.2 Implementing a Split Radix FFT

For this implementation, the smallest transform sizes are eight and four (the 2-input DFTs are integrated into the 8-input and 4-input transforms). These sizes allow the trivial multiplications to be hard-coded for efficiency. Table 2 shows the flow of a 128-point split-radix FFT.

**Table 2: Split Diagram for the FFT - 128 points**

| | | | | |
|---|---|---|---|---|
| 128 | 64 | 32 | 16 | 8 |
| | | | | 4 |
| | | | | 4 |
| | | | 8 | |
| | | | 8 | |
| | | 16 | 8 | |
| | | | 4 | |
| | | | 4 | |
| | | 16 | 8 | |
| | | | 4 | |
| | | | 4 | |
| | 32 | 16 | 8 | |
| | | | 4 | |
| | | | 4 | |
| | | 8 | | |
| | | 8 | | |
| | 32 | 16 | 8 | |
| | | | 4 | |
| | | | 4 | |
| | | 8 | | |
| | | 8 | | |

# 3 Conclusion

Streaming SIMD Extensions provide increased performance for the FFT algorithm. The main reason for the performance gain is SIMD floating point (as opposed to x87 scalar) operations. The advantage to using Streaming SIMD Extensions is the increase in precision over MMX technology.

When both speed and accuracy are a concern, it is likely that the Streaming SIMD Extensions implementation will be the best solution.

# 4 Coding Examples

The *samples* directory contains the code for the algorithms described here. The following files can be found there, corresponding to the given implementations:

| Implementation: | File Name: |
|---|---|
| C++ vector classes | srfftm2v.cpp |
| Intrinsics | srfftm2i.cpp |
| Assembly | srfftm2a.asm |

A Microsoft Visual C++ 5.0 project file and a MS-DOS batch file are included, either of which can be used to build a test program that invokes these implementations.